CS-GY 9223: Deep Learning: Neural Fingerprinter

Jason Woo
Computer Science and Engineering
New York University Tandon School of Engineering
Brooklyn, NY 11201
jwoo@nyu.edu

December 15, 2020

Abstract

You are in your favorite bar, coffee shop, or wherever and you hear a song that captures your interest. Before, you would have ask whoever is playing the music for the track title. Understanding a technological solution, Avery Wang, co founder of Shazam, wrote the paper: An Industrial-Strength Audio Search Algorithm that algorithmically solves the problem using Locality Sensitive Hashing (LSH) and fingerprinting. Shazam is extremely common and its API is even embedded into every iPhone. All you need to say is "hey Siri, what song is this?" and Shazam will identify the song. While LSH is extremely accurate, there are computation limitations. The noisy recorded audio clip from your phone is sent to Shazam and they do the required computations. Google, with their own line of phones, proposed a different solution using neural networks. This report discusses an implementation inspired by Googles paper: Now Playing: Continuous low-power music recognition.

1 Introduction

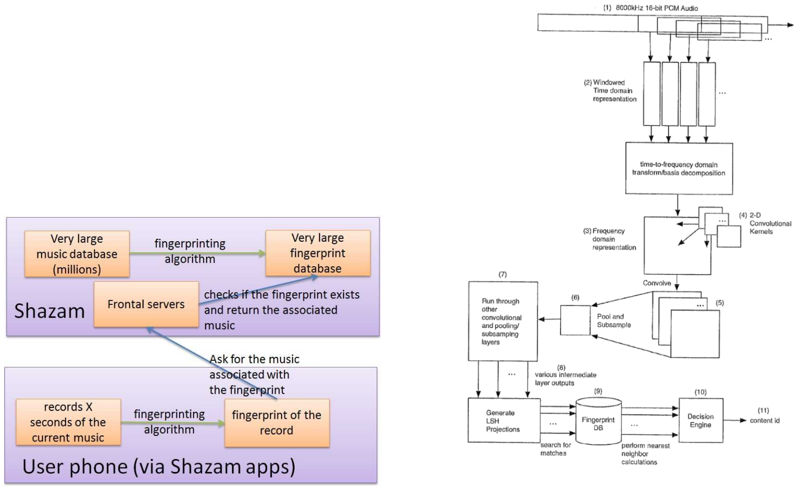
Shazam's Efforts using LSH Shazam and competitors, such as Sound Hound, using very similar architectures that involve fingerprinting and LSH. These companies take in audio data and compress it to find its fingerprint. The fingerprint is a sketch of the original audio input such that the fingerprint preserve properties of the original data. In the case for Shazam, they need to preserve Jaccard Similarity scores (JS scores), such that the Jaccard Similarity of fingerprints of songs are very similar to the JS scores of the original data. There is some preprocessing done on the total database of full length songs, to find all of their fingerprints as well

Google's Efforts using Neural Networks The problem with fingerprinting using LSH is that it is relatively computationally expensive, so audio signals recorded on a users device must be sent to Shazam's servers to do calculate its fingerprint and similarity score. An intuitive solution to this is to use auto encoding neural networks. The reason why this is a more tenable solution, in terms of per use computation, is that given a saved model with pre-trained weights, a phone can locally run audio clips to this pre-trained models, instead of needing to send it to Google's servers to do the computation. Like the ubiquitous word2vec algorithm, the goal is to compress the data and preserve valuable information. Google researched and implemented a solution to this problem using neural networks. A diagram of their architecture from their patent filing is shown below.

2 Literature Survey

Survey Paper Aside from the original white paper written by Avery Wang for Shazam, there is a sizable number of papers discussing audio fingerprinting using traditional methods, such as LSH, which is shown in the survey paper: Survey and Evaluation of Audio Fingerprinting Schemes. It summarizes three papers regarding the topic. All three papers follow the same general structure to creating the fingerprints, shown below.

Google's Paper However, there is not much published research in methods using neural networks to learn optimal fingerprints. Google released a paper in 2017 that outlines a network architecture to do audio fingerprinting, called


Very large music database (millions)
fingerprinting algorithm
Very large fingerprint database
Shazam
Frontal servers
checks if the fingerprint exists and return the associated music
Ask for the music associated with the fingerprint
records X seconds of the current music
fingerprinting algorithm
fingerprint of the record
User phone (via Shazam apps)
(1) 8000kHz 16-bit PCM Audio
(2) Windowed Time domain representation
time-to-frequency domain transform/basis decomposition
(3) Frequency domain representation
(4) 2-D Convolutional Kernels
Convolve
(5)
(6)
Pool and Subsample
(7)
Run through other convolutional and pooling/ subsamping layers
(8) various intermediate layer outputs
(9)
Generate LSH Projections
Fingerprint DB
search for matches
perform nearest neighbor calculations
(10)
Decision Engine
(11) content id

(a) Shazam (b) Google

Figure 1: Shazam and Google's architectures

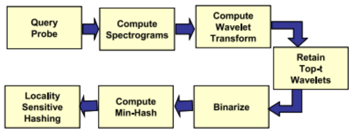


Figure 2: Survey Paper Architecture

Neural Network Fingerprinter (NNFP). The motivator for Google to develop an alternative solution to fingerprinting is because traditional methods send data to a server for computation, but Google wanted a method to be run solely on a mobile device. They needed a low power/computation solution to the problem.

The only other paper I can find regarding neural networks for fingerprinting of audio data was a paper published by Korean researchers at Cochlear.ai, Seoul National University and SK Telecom: Neural Audio Fingerprint for High-Specific Audio Retrieval Based on Contrastive Learning. This paper was released after Google's paper with the main modification being the loss function used which results in impressive gains. "In their [Google's] setup, Now-playing could identify songs within 44h long audio database. In our benchmark, we replicate this semi-hard triplet approach and compare it with our work in a new setup: high-specific audio retrieval in a 180 times larger database."

Follow up to Google's Paper I found this followup paper particularly helpful because they provide a diagram that very clearly outlines what is happening.

They calculate overlapping segment-wise log Mel-spectrograms, each being 2 seconds long. They then enter all these spectrograms into a convolutions encoder, and then ultimately into a L2 projection layer. What is outputted is a series of fingerprints each for a segment of a song. They don't mention the size of the reduced dimension of the fingerprints, but that makes sense because this is a trial and error process to see which embedding size most accurately retains the information.

3 Data Set

Fortunately, music data is one of the most common and ubiquitous forms of data on the internet. Unfortunately, it is heavily copyrighted and difficult to find large data sets! Through some research, I found several websites that have aggregated databases of downloadable songs under the creative commons license. Some of them being AcousticBrainz, FMA and AudioSet. They all provide MP3 files, which would need to converted to WAV format files, so common packages in Python can create spectrograms out of the files. This can easily be done with a simple script. The query sound clips will be manually uploaded. I will record a noisy version of the song with my phone, including some background sound, and use that as a query.

The data set I settled on using for this project is the FMA (For Music Analysis) data set. FMA: A Data set for Music Analysis, is the associated paper. I decided on using this data set for several reasons. First, its extremely easy to download their zipped files using wget. Second, they have full and abridged data. I originally attempted to using their full data set of over 106,500 complete songs, but realized due to computation limitation that I needed a shorter version of their data, not just in terms of number of songs, but also length of songs, which they provide. The size of their data I chose to use is 8000 songs, each being a 30 second clip of the original full length song.

4 Model Selection

The model I will use for this project is a combination of methods proposed by Google in, “Now Playing: Continuous low-power music recognition” as well as "Neural Audio Fingerprint for High-Specific Audio Retrieval Based on Contrastive Learning" by Chang, Lee, et al. A diagram of the architecture proposed by Google is shown below, but I’ll also provide some commentary. A few seconds of audio is used as a query, however, an embedding is produced at a rate of one embedded vector per second of query. So the input to NNFP is a second of an audio clip. The one second audio clip, which is just a long vector of numbers, is initially passed to a “stack of convolutional layers.” The paper doesn’t explicitly say the number of convolutional layers, so I will experiment with this, however, their diagram portrays many convolutions, more than 6. After the series of convolutional layers, the data is passed to a two level “divide-and-encode layer.” The details of that implementation is referenced in this paper. The loss function used is the “triplet loss function” which its outlined in this paper. “The NNFP model is trained on a data set of noisy music segments which are aligned to the corresponding segment in their reference song."

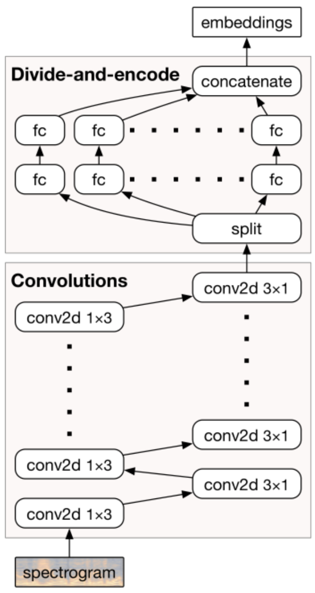


Figure 3: Google NNFP

I will combine the two methods proposed by Google and Cochlear.ai. While Google uses CNNs followed by a divide-and-encode block, I will not use a divide-and-encode blocks. Instead I will do as Chang, Lee, et al and use a fully connected L2 projection layer after using CNNs. The reason for this modification is because of complexity. As Cochlear.ai shows, they achieved more generalizable results without the need for a complicated divide-and-encode block, which would increase the number of trainable paramaters. However, I will not use the loss function proposed by Chang, Lee, et al. and instead use the triple loss function as described by Google. The reason for this is for simple curiosity.

We have not discussed the triplet loss function in class, so I will provide some details about it here. According to the paper, by Schroff, Kalenichenko and Philbin at Google: FaceNet: A Unified Embedding for Face Recognition and Clustering, the triplet loss function "minimizes the distance between and anchor and a positive both of which have the same identity, and maximizes the distance between the anchor and a negative of a different identity."[6] So, specifically in the context of the audio fingerprinting problem, the triplet loss function ensures that "every audio segment minimizes the distance to examples of the same audio segment, while ensuring that their distances to all other audio segments are larger." [2] I've checked both Tensorflow and Pytorch, the former provides the function built in while in Pytorch there are implementations of it floating around, as linked above.

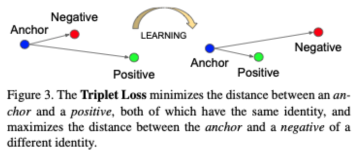


Figure 3. The **Triplet Loss** minimizes the distance between an *anchor* and a *positive*, both of which have the same identity, and maximizes the distance between the *anchor* and a *negative* of a different identity.

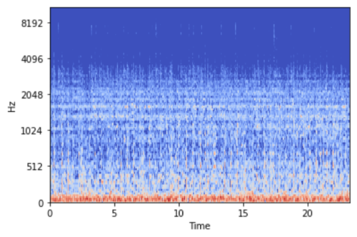
Figure 4: Triplet Loss Function

In this particular project, the anchor would be the log mel spectrogram, the positive will be a noisy version of the log mel spectrogram and the negative will be a random element from the original log mel spectrogram list. While TensorFlow offers a version of the triplet loss function, I decide to implement my own version, because their documentation was quite confusing to define my own network and then to pass the required anchor, positive and negative triplet to their loss function.

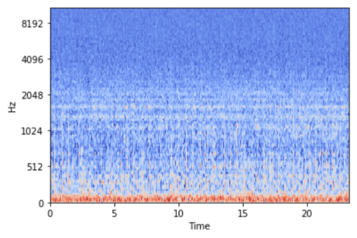
5 Implementation

In this section I will provide some code and discuss specific implementation decisions I made. I began the project by deciding to use NYU's HPC environment for development. This by itself provides several technical challenges, such as becoming familiar with sbatch and sending work to computation nodes. In the provided github repository, I provide the sbatch scripts and the bash commands to be able to open the jupyter note books. I also provide two conda environments, one for the cleaning notebook and one for the notebook dedicated for model creation and training. Details for how to use the code is provided in the repository's README.

Aside from developing environment, a key decision I needed to make is how to create the positive log mel spectrograms for the triplet loss function. The triplet loss function is often used in the computer vision setting, with the positive node being a part of the same class, if its a classification setting. So, for example in MNIST, a positive node would be a different image with the same class as the anchor. In my project, there are no classes to rely on, so to overcome this I added Additive White Gaussian Noise (AWGN) to the original signal and then calculated the log mel spectrogram on that to produce the positive nodes. The referenced a Medium article: Adding noise to audio clips to create the functions necessary to create Gaussian noise. Examples of the original and noisy log mel spectrograms are shown below.



(a) original



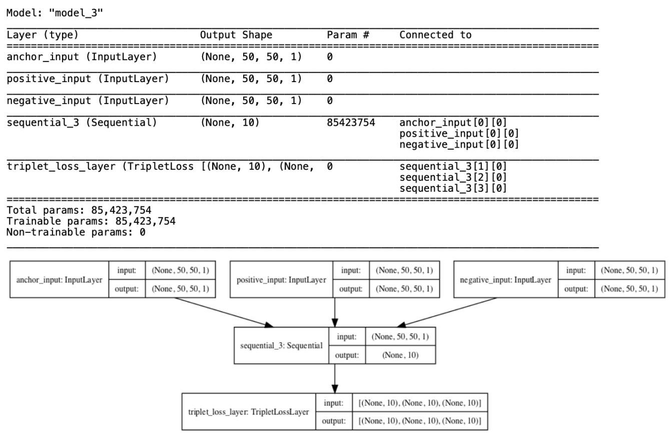
(b) noisy

Figure 5: Original and Noisy Log Mel Spectrograms

After looping through the FMA data and converting each song to their log mel spectrograms, I noticed the size of the spectrograms is roughly 128x2000. While networks have been developed for images this large, I did not want to harp on the computational complexity for this project. My solution is simple, simply randomly sample 50 rows and columns to create an smaller input. I tested this on two different networks, one with input size of

50x50 and another with input size 128x128, still had to reduce the data size because of computation purposes, and I saw similar losses, off by $\frac{1}{1000}$. This was justification enough to stick with randomly sampling the log mel spectrogram.

Next, I had to create the model. The full code is provided in the attached github, but I provide a diagram of the model below. The key take away from the diagram is that the model is taking in three inputs, anchor, positive and negative nodes, passing it to a keras model, then passing the three values to a triplet loss function calculator.



```
Model: "model_3"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
anchor_input (InputLayer)       (None, 50, 50, 1)    0
__________________________________________________________________________________________________
positive_input (InputLayer)     (None, 50, 50, 1)    0
__________________________________________________________________________________________________
negative_input (InputLayer)     (None, 50, 50, 1)    0
__________________________________________________________________________________________________
sequential_3 (Sequential)       (None, 10)           85423754    anchor_input[0][0]
                                                                 positive_input[0][0]
                                                                 negative_input[0][0]
__________________________________________________________________________________________________
triplet_loss_layer (TripletLoss [(None, 10), (None,  0           sequential_3[1][0]
                                                                 sequential_3[2][0]
                                                                 sequential_3[3][0]
==================================================================================================
Total params: 85,423,754
Trainable params: 85,423,754
Non-trainable params: 0
__________________________________________________________________________________________________
```

anchor_input: InputLayer
input: (None, 50, 50, 1)
output: (None, 50, 50, 1)
positive_input: InputLayer
input: (None, 50, 50, 1)
output: (None, 50, 50, 1)
negative_input: InputLayer
input: (None, 50, 50, 1)
output: (None, 50, 50, 1)
sequential_3: Sequential
input: (None, 50, 50, 1)
output: (None, 10)
triplet_loss_layer: TripletLossLayer
input: [(None, 10), (None, 10), (None, 10)]
output: [(None, 10), (None, 10), (None, 10)]

Figure 6: Neural Network Diagram

A complete outline of the model is provided below and it is heavily adapted from this github repository which was used to do classification on MNIST. I adapted it to take in log mel spectograms, instead of MNIST examples. The model takes in, as input, a batch of data that has the three required elements for the triplet loss function. I decided to create mini-batches, each with 50 examples of the randomly sampled log mel spectrograms. When I trained the model, I chose 5 epochs, each being trained on 140 mini-batches.

The average training loss per epoch is provided below.

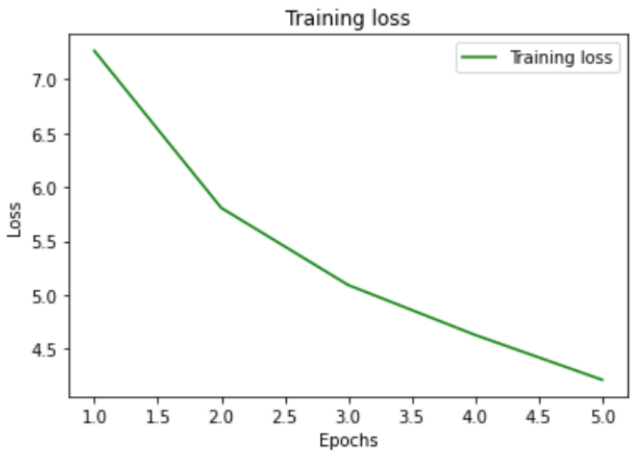


Figure 7: Training Loss Function

Because this is an embedding problem with out any class associated for the songs, there is no training accuracy. It is also important to highlight the importance of embedding size. Neither papers by Google or Chang, Lee, et al. provide a embedding size parameter, so I chose a very small embedding size of 10. While, in practice, this would be too small to compare a song with potentially thousands of other potential matches, I decided to use 10 because I am mostly concerned in implementing the network, not in finding the optimal embedding.

```
def build_network(input_shape, embeddingsize):
    # Convolutional Neural Network
    network = Sequential()
    network.add(Conv2D(128, (3,3), activation='relu',
                     input_shape=input_shape,
                     kernel_initializer='he_uniform',
                     kernel_regularizer=l2(2e-4)))
    network.add(MaxPooling2D())
    network.add(Conv2D(128, (3,3), activation='relu', kernel_initializer='he_uniform',
                     kernel_regularizer=l2(2e-4)))
    network.add(MaxPooling2D())
    network.add(Conv2D(256, (3,3), activation='relu', kernel_initializer='he_uniform',
                     kernel_regularizer=l2(2e-4)))
    network.add(Flatten())
    network.add(Dense(4096, activation='relu',
                   kernel_regularizer=l2(1e-3),
                   kernel_initializer='he_uniform'))
    network.add(Dense(embeddingsize, activation=None,
                   kernel_regularizer=l2(1e-3),
                  kernel_initializer='he_uniform'))

    #Force the encoding to live on the d-dimentional hypershpere
    network.add(Lambda(lambda x: K.l2_normalize(x,axis=-1)))

    return network

class TripletLossLayer(Layer):
    def __init__(self, alpha, **kwargs):
        self.alpha = alpha
        super(TripletLossLayer, self).__init__(**kwargs)

    def triplet_loss(self, inputs):
        anchor, positive, negative = inputs
        p_dist = K.sum(K.square(anchor-positive), axis=-1)
        n_dist = K.sum(K.square(anchor-negative), axis=-1)
        return K.sum(K.maximum(p_dist - n_dist + self.alpha, 0), axis=0)

    def call(self, inputs):
        loss = self.triplet_loss(inputs)
        self.add_loss(loss)
        return loss

def build_model(input_shape, network, margin=0.2):
    # Define the tensors for the three input images
    anchor_input = Input(input_shape, name="anchor_input")
    positive_input = Input(input_shape, name="positive_input")
    negative_input = Input(input_shape, name="negative_input")

    # Generate the encodings (feature vectors) for the three images
    encoded_a = network(anchor_input)
    encoded_p = network(positive_input)
    encoded_n = network(negative_input)

    #TripletLoss Layer
    loss_layer =
        TripletLossLayer(alpha=margin,name='triplet_loss_layer')([encoded_a,encoded_p,encoded_n])

    # Connect the inputs with the outputs
    network_train = Model(inputs=[anchor_input,positive_input,negative_input],outputs=loss_layer)

    # return the model
    return network_train
```

6 Conclusion

Intended Beginning this project, I wanted to create small end-to-end version of what Google suggested in their paper, a system that uses a neural network to fingerprint a small audio clip to compare to a larger database of fingerprints, and ultimately output a potential match. However, when actually implementing the system, I realized that much of the scope of that was too large and didn't focus on neural networks, it used, rather, fingerprinting as a small, but crucial aspect of a larger system.

Accomplished So, the final deliverable I have is instead, a neural network that takes as input a 30 second long clip of music, converts it to its log mel spectrogram, randomly sample from this spectrogram, send it to a neural network using a triplet loss function, and outputs a compressed version of the inputted audio clip.

Challenges The main challenge of this project was computational complexity. The original size of the log mel spectrogram was too large to run the network efficiently given my level of understanding. If I continue developing this project, which I hope I will because it truly is quite fun, I will want to spend more time understanding GPU usage with sbatch. I understand how to use GPUs with TensorFlow in Google's colab environment, but I couldn't figure it out in time for this project using NYU's HPC resources. I had a consolation with someone from NYU's HPC team, and I would need to spend significantly more time to develop something to use GPUs on a distributed system.

Another challenge I had was the way I made the negative node for the triplet loss function. The way I did so was to randomly shuffle the anchor list and hope that this would result in a far enough output. However, this is not the most efficient way. I more intuitive approach is to have genre classes for songs and pick the negative node to be in a very different genre. For time constraint purposes, I did not do this.

The final GitHub repository for this project can be found here.

References

[1] S. Baluja and M. Covell. Content fingerprinting using wavelets. In Proc. of Euro- pean Conference on Visual Media Production (CVMP), London, UK, Nov, 2006.

[2] B. Agüera, B. Gfeller, R. Guo, K. Kilgour, S. Kumar, J. Lyon, J. Odell, M. Ritter, D. Roblek, M. Sharifi, M. Velimirovic . Now Playing: Continuous low-power music recognition

[3] V. Chandrasekhar, M. Sharifi, D. Ross. Survey and Evaluation of Audio Fingerprinting Schemes for Mobile Query-by-Example Applications

[4] H. Lai, Y. Pan, Y. Liu, and S. Yan. Simultaneous feature learning and hash coding with deep neural networks. In The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), June 2015.

[5] D.-A. Clevert, T. Unterthiner, and S. Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). arXiv preprint arXiv:1511.07289, 2015.

[6] F. Schroff, D. Kalenichenko, and J. Philbin. Facenet: A unified embedding for face recognition and clustering. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 815–823, 2015.

[7] Defferrard, Michaël Benzi, Kirell Vandergheynst, Pierre Bresson, Xavier. (2016). FMA: A Dataset For Music Analysis.

[8] Sungkyun Chang, Donmoon Lee, Jeongsoo Park, Hyungui Lim, Kyogu Lee, Karam Ko, Yoonchang Han: Neural Audio Fingerprint for High-specific Audio Retrieval based on Contrastive Learning. CoRR abs/2010.11910 (2020)